

Security Lab – Developing Secure Web Applications and RESTful Web Services: Extending Marketplace

Virtual Machine

This lab can be done with the **Ubuntu VM**. The remainder of this document assumes you are working with this VM.

You can basically also do this lab on your own system, but several software packages have to be installed (IDE, Java, Payara, MySQL,...). All this is already installed on the Ubuntu VM, so it's easiest to work with this VM.

1 Introduction

In this lab, you are extending the Marketplace application from the lecture. The goal of the lab is that you get more experienced with developing secure web applications and RESTful web services by using the security features provided by Jakarta EE and by designing and implementing your own security functions. The lab is based on the final version from the lecture, and it is assumed that you are familiar quite well with that version.

2 Basis for this Lab

- Download the following files from Moodle:
 - *Marketplace_Lab.zip*
 - *Marketplace.sql* and *Marketplace_UpdateEncryptedCreditCards.sql*
- Move the files to an appropriate location (e.g., into a directory *securitylabs* in the home directory */home/user*).
- To create the database schema and the technical user used by the Marketplace application to access the database, do the following (this can be repeated at any time to reset the database):
 - Open *MySQL Workbench*.
 - Click on the left on *Local Instance 3306* and enter *root* as password.
 - Choose *Open SQL Script...* in the menu *File* and select the downloaded file *Marketplace.sql*.
 - Click the *Execute* icon.
- Unzip the downloaded zip file. This results in a directory *Marketplace_Lab* that contains four directories *Marketplace_Lab-common*, *Marketplace_Lab-web*, *Marketplace_Lab-rest* and *Marketplace_Lab-rest-client*. The first three directories contain three Jakarta EE projects and provide the Marketplace web application and the Marketplace RESTful web service. The fourth directory contains a Java project that is used to test the web service. All projects are based on Maven and should be importable in any modern IDE. The remainder assumes you are using *NetBeans*, which is installed on the Ubuntu image.
- Start *NetBeans* and open all four projects.
- To build either the web application or the web service, first right-click *Marketplace_Lab-common* (as this contains common functionality for both the web application and the web service) and select *Clean and Build*. Then, right-click either *Marketplace_Lab-web* or *Marketplace_Lab-rest* and select *Clean and Build*.
- To build *Marketplace_Lab-rest-client*, right-click it and select *Build*.
- To run the web application or the web service, do the following:
 - To run the web application, right-click *Marketplace_Lab-web* and select *Run*. If Payara (which is the Jakarta EE application server that is used in this lab to run the Webshop appli-

- cation) is not yet running, it will be started, which takes some time. The web application will be used from the beginning of this lab.
- To run the RESTful web service, right-click *Marketplace_Lab-rest* and select *Run*. If Payara is not yet running, it will be started, which takes some time. The web service will be used later in this lab (task 4, see Section 9).
 - Whenever you do some changes to *Marketplace_Lab-common*, *Marketplace_Lab-web* or *Marketplace_Lab-rest*, use *Clean and Build* and *Run* again (as described above) before testing the updated version. This should ensure that the fully updated version is deployed on Payara.
 - To run *Marketplace_Lab-rest-client*, right-click *Test.java* in package *ch.zhaw.securitylab.marketplace.rest.test* and select *Run File*. This will also be used later in this lab (task 4, see Section 9).
 - Under the *Service* tab and expanding *Servers*, the Payara server is listed. Sometimes, it may be necessary to restart Payara, which can be done with a right-click and selecting *Restart*. Under *Applications* (below the Payara server), you can also undeploy applications if necessary.
 - The web application is reached with https://localhost:8181/Marketplace_Lab-web. Ignore the certificate warning you'll get when accessing it, as this is only a testing environment. You can also accept the certificate permanently to get rid of the warning.
 - The RESTful web service is reached with URLs below https://localhost:8181/Marketplace_Lab-rest/rest/, e.g., https://localhost:8181/Marketplace_Lab-rest/rest/products.

3 Project Organization

As you have seen above, the Marketplace web application and web service consist of multiple projects. The main reason for using multiple projects is that we are using different *web.xml* configuration files and different ways of authentication for the web application and the web service, and this can easiest be done by using multiple projects. The three projects are organized as follows:

- *Marketplace_Lab-common*: Contains the common classes that are used both by the web application and the web service. This includes the entity classes, the database facades, service classes, validation classes, utility classes, and data transfer object (DTO) classes.
- *Marketplace_Lab-web*: Contains everything specific to the Jakarta Faces web application. This includes the Facelets, the backing bean classes, the web application-specific security classes and the web application-specific *web.xml*.
- *Marketplace_Lab-rest*: Contains everything specific to the RESTful web service. This includes the classes to handle RESTful communication, the web service-specific security classes the web service-specific *web.xml*.

4 General Remarks

Your primary objective is to solve the four tasks in Sections 6 - 9. In the first three tasks, you'll extend the Marketplace web application and in the fourth task, you'll extend the Marketplace web service.

Beyond solving the tasks correctly, it is also expected that your extensions do not introduce typical vulnerabilities such as Cross-Site Scripting, SQL injection, access control problems and so on. As you know from the lecture, this can be prevented by correctly using the security features that are offered by Jakarta EE.

Make sure to study the provided skeletons of Facelets and Java classes so you understand them before you implement your extensions.

The Payara log includes, among other information, possible exceptions that are thrown within the application, which may be helpful during debugging of your program extensions.

Finally, the lab points section at the end contains the tests your program has to pass to get the lab points. When you have solved a part, it's strongly recommended you check whether the corresponding tests work as expected. If anything doesn't work, you should first fix the problem before continuing.

5 Users, Passwords, and Roles

The two following tables list the available users, their passwords, and the roles they get after successful login. For instance, user *alice* has the password *rabbit* and gets the role *sales* after successful login. The passwords are shown here in plaintext, but they are stored securely in the database (using PBKDF2, as discussed in the lecture).

Username	Password	Username	Rolename
alice	rabbit	alice	sales
bob	patrick	bob	burgerman
donald	daisy	donald	productmanager
john	wildwest	john	sales
luke	force	luke	productmanager
robin	arrow	robin	marketing
snoopy	woodstock	snoopy	productmanager

6 Task 1: Extending Admin Area

Your first task is to extend the admin area of the web application so that products can be listed, removed, and added. This is only allowed by users with the role *productmanager*. In addition, every authenticated user (with any role) should be able to change his password.

6.1 Listing and Removing Products

In the first step, *view/admin/admin.xhtml* should be extended such that the products are listed if a user with role *productmanager* accesses the admin area. Of course, this requires user authentication if the user is not logged in, but this functionality was already implemented during the lecture and is therefore included in the version you are using.

Depending on the role of the user, the admin area should appear as follows:

- Role *sales* (role *marketing* is similar, but does not display the *Remove purchase* buttons):

Admin Area

Purchases:

First Name	Last Name	Credit Card Number	Total Price (CHF)	
Ferrari	Driver	1111 2222 3333 4444	250000.00	Remove purchase
C64	Freak	1234 5678 9012 3456	444.95	Remove purchase
Script	Lover	5555 6666 7777 8888	10.95	Remove purchase

[Return to search page](#) [Account settings](#) [Logout](#)

- Role *productmanager*:

Admin Area				
Products:				
Code	Description	Price (CHF)	Username	
0001	DVD Life of Brian - used, some scratches but still works	5,95	donald	<input type="button" value="Remove product"/>
0002	Ferrari F50 - red, 43000 km, no accidents	250000,00	luke	
0003	Commodore C64 - used, the best computer ever built	444,95	luke	
0004	Printed Software-Security script - brand new	10,95	donald	<input type="button" value="Remove product"/>
<input type="button" value="Return to search page"/> <input type="button" value="Add product"/> <input type="button" value="Account settings"/> <input type="button" value="Logout"/>				

- Role *burgerman* (any other role than *marketing* / *sales* / *productmanager*):

Admin Area	
Your role does not provide access to special functions in the admin area.	
<input type="button" value="Return to search page"/> <input type="button" value="Account settings"/> <input type="button" value="Logout"/>	

Perform the following steps to implement this functionality:

- Add *security-role* elements to *web.xml* (in *Marketplace_Lab-web*) for the two new roles *productmanager* and *burgerman* and then adapt *web.xml* such that the two new roles also have access to resources below *view/admin/*.
- Extend *view/admin/admin.xhtml*. The version from the lecture has already been adapted such that for roles *sales*, *marketing* and *burgerman*, the correct content is displayed. You must extend *admin.xhtml* such that it also shows the correct content for role *productmanager*. The element `<h:panelGroup>` for the roles *sales* and *marketing* should provide you with a good template how this can be done. As the backing bean, you can use *AdminProductBacking.java*, which is already completely implemented.
- Make sure that a product manager is only allowed to *delete the own products*. For instance, in the example above, *donald* is logged in and he should only be able to delete the products with codes 0001 and 0004, where his name is listed in the *Username* column of the table. The best way to do this with Jakarta Faces is to include (using the *rendered* attribute in the `<h:form>` tag) the *Remove product* button only in the rows that can be deleted by the current user, as shown above (within the Facelet, you can get the name of the current user with *request.getRemoteUser()*).

Before you continue, check whether the applicable tests in the lab points section work as expected.

6.2 Adding new Products

Clicking the *Add product* button in the admin area allows product managers to add new products. The following must be true such that a new product is accepted:

- The code must contain exactly 4 characters (lowercase letters, uppercase letters, or digits).
- The description must contain at least 10 and at most 100 characters. Allowed characters are letters (lower- and uppercase), digits, the space character, and the special characters comma (,), quote (') and dash (-).
- The price must be a decimal number in the range 0 – 999'999.99, with at most two digits after the decimal point.
- In addition, the code must be different from all codes of already existing products.

The behavior should be as follows:

- Clicking the *Add product* button opens the Facelet view `admin/product/addproduct.xhtml`.

Add Product

Please provide the following information to add a product:

Code:

Description:

Price:

- Entering invalid data for the input fields and clicking *Add product* result in validation errors being displayed:

Add Product

Please provide the following information to add a product:

Code: *Please insert a valid code (4 letters / digits)*

Description: *Please insert a valid description (10-100 characters: letters / digits / - / , / ')*

Price: *Please insert a valid price (between 0 and 999'999.99, with at most two decimal places)*

- Entering an already existing code also displays an error:

Add Product

The product could not be added as a product with the same code already exists

Please provide the following information to add a product:

Code:

Description:

Price:

- Entering valid data and clicking *Add product* adds the product and returns to the admin area:

Admin Area

The product could successfully be added

Marketplace products:

Code	Description	Price (CHF)	Username	
0001	DVD Life of Brian - used, some scratches but still works	5.95	donald	<input type="button" value="Remove product"/>
0002	Ferrari F50 - red, 43000 km, no accidents	250000.00	luke	
0003	Commodore C64 - used, the best computer ever built	444.95	luke	
0004	Printed Software-Security script - brand new	10.95	donald	<input type="button" value="Remove product"/>
0005	Super Vulnerability Scanner	9999.95	donald	<input type="button" value="Remove product"/>

Perform the following steps to implement this functionality:

- Adapt `web.xml` such that only role `productmanager` can access resources below `view/admin/product/` and only over HTTPS. You can do this with an additional `<security-constraint>` and without modifying the existing `<security-constraint>` for the admin area (i.e., the one for URLs below `view/admin/`).

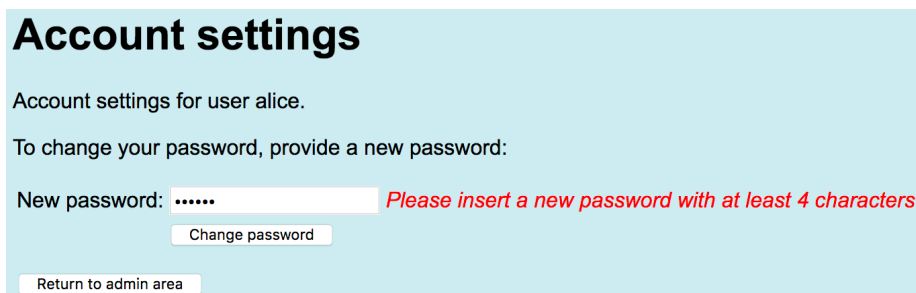
Note that with this additional `<security-constraint>` you just added, it is no longer possible that other roles than *productmanager* can access resources below *view/admin/product/* (which includes *addproduct.xhtml*), although – according to the already existing `<security-constraint>` for the admin area – this seems to be still possible, as URLs below *view/admin/product/* are also below *view/admin/*. The reason why access is no longer possible is because the `<url-pattern>` defined in the new `<security-constraint>` has a better match when resources below *view/admin/product* are accessed, i.e., the `<url-pattern>` */view/admin/product/** matches better than */view/admin/**. Therefore, the only newly added `<security-constraint>` will be evaluated when accessing resources below *view/admin/product/* and consequently, only *productmanager* gets access. So, the general rule when Jakarta EE evaluates `<security-constraint>`s is that it always considers only the `<security-constraint>` that has the best match according to the `<url-pattern>`.

- Extend *view/admin/product/addproduct.xhtml*. This should be done similar as in *view/public/secure/checkout.xhtml*. As the backing bean, you can use *AddProductBacking.java*, which is already completely implemented. By inspecting this bean, you can see in method *addProduct* how the name of the current user is determined and set in the *Product* entity before the entity is inserted into the database.
- To actually insert the *Product* entity into the database, method *addProduct* in *AddProductBacking.java* uses the method *insertProduct* in *AdminProductService.java*. You must implement this method. The method has to check whether the product code of the new product already exists. If this is the case, return false and don't insert the *Product* entity into the database. Otherwise, insert it and return true. The reason for placing this functionality in a separate service class is because it will also be used by the web service later. Note that to check whether a product with a specific product code already exists in the database, you can use the method *findByCode* in *ProductFacade.java*.
- Add Bean Validation annotations¹ to the instance variables *code*, *description* and *price* in *Product.java* to enforce the restrictions described above and to make sure that only valid products are inserted into the database (note that guaranteeing the uniqueness of the product code including displaying the correct error message in case the code is not unique should already be done correctly in method *addProduct* in *AddProductBacking.java*, assuming you have correctly implemented method *insertProduct* in *AdminProductService.java*). Using the Bean Validation annotations is done similar as in *Purchase.java*. The *code* and *description* can be handled with `@Pattern`. For the *price*, it's best to combine `@NotNull`, `@PositiveOrZero` and `@Digits` (simply use all of them in front of the attribute *price*, which enforces them all). Make sure to use the same validation error messages as illustrated in the screenshot.

Before you continue, check whether the applicable tests in the lab points section work as expected.

6.3 Account Settings – Change Password

In the admin area, there's a button *Account settings*. Clicking this button opens the Facelet *view/admin/account/accountsettings.xhtml* as illustrated below:



¹Details about the annotations: <https://eclipse-ee4j.github.io/jakartaee-tutorial/#bean-validation>

The page allows users to change their password. The Facelet uses *AccountSettingsBacking.java* as backing bean, which itself uses method *changePassword* of *AccountSettingsService.java* to perform the actual password change. Note that there are minimal requirements for the new password (at least 4 characters), which is of course not reasonable to be used in practice but which we consider good enough for this lab setting (feel free to increase the password requirements if you like).

Your task is to complete the functionality to change the password. Perform the following steps to implement this functionality:

- Looking at the screenshot above, you should realize that this password change is designed in an insecure way (and with this, we don't mean the too simplistic requirements for the new password, and also not the fact the user doesn't have to insert the new password twice (which would be reasonable to do, but which is omitted here)). Identify the issue and start fixing it by adapting the Facelet, the backing bean, and the parameter list of method *changePassword* in *AccountSettingsService.java*.
- Next, adapt the code in method *changePassword* in *AccountSettingsService.java* so that it allows to change the password in a secure way. If the password change is successful, the method returns true, otherwise false. Take the following into account when adapting the method:
 - The method already contains the code to get the correct *UserInfo* entity from the database and to update the modified entity in the database (so the new PBKDF2 hash corresponding to the new password is stored).
 - Within *changePassword*, use an object of type *Pbkdf2PasswordHash* (it's already injected in *AccountSettingsService.java*), which provides methods to verify passwords and to generate new PBKDF2 hashes (details see Jakarta EE API documentation). Note that before generating a new PBKDF2 hash, the *Pbkdf2PasswordHash* object must be initialized so it uses the desired number of iterations (100'000), HMAC function (PBKDF2WithHmacSHA512), salt size (64 bytes) and hash size (32 bytes). The corresponding code of this initialization is also already available in method *changePassword*.
- Finally, think about whether you have to adapt *web.xml* to correctly configure access control for the account settings. If yes, add the required configurations. Explain whether you have to add anything or not.



Before you continue, check whether the applicable tests in the lab points section work as expected.

6.4 Access Control Considerations

Before you continue to extend the application further, you should have a more detailed look at access control. In Section 6.2, you used the *rendered* attribute to make sure that the *Remove product* button only shows up in the rows that contain products of the current user. The question is now whether this is truly enough to enforce that a product manager can only delete the own products. If this is not the case, then we would have a broken object level access control vulnerability.

To analyze this, look at the HTML code of the admin area page when it is displayed in the browser. If you analyze the *<form>* tags that correspond to the *Remove product* buttons, you can see that every form uses slightly different names and values for some parameters of the form. E.g., *j_idt35:0:j_idt49* is used in the 1st row, *j_idt35:3:j_idt49*, is used in the 4th row etc., so it seems that the numbers in bold identify the row of the table (starting with 0 for the 1st row). The question is now whether a malicious product manager can use the names and values «of other rows» to delete products of other product managers.

To test this, start an interceptor proxy. On the image, you can use *Burp Suite* for this, which is already installed. It can be started via the left-side menu bar. After startup, select *Temporary Project* on the first screen and *Load from Configuration File /securitylab/burpsuite/burp.json* (you must mark the file) on the second screen. This makes sure the proxy listens on port 8008. Next, configure the browser to use *Burp Suite* as a proxy. For this, assuming you are working with the image, open the *Settings* in Firefox, select *General*, then scroll down to *Network Settings* and click *Settings*. In the opening window, select *Manual proxy configuration* and enter *localhost* and port *8008* under *HTTP proxy*, and mark the checkbox *Also use this proxy for HTTPS*. In addition, remove all entries from the *No Proxy for* box in case there are entries listed. As the browser and the proxy (*Burp Suite*) are running on the same system, you need to change another Firefox setting to make sure that requests to *localhost* are actually sent through *Burp Suite*. To do this, enter *about:config* in the address bar and then click on the button *Accept the Risk and Continue*. Then, enter *localhost* in the search field and double-click on the line *network.proxy.allow_hijacking_localhost* to set it to *true*.

Next, go back to the browser and click the *Remove product* button in one of the rows. In *Burp Suite*, modify the POST parameters of the resulting request so that they correspond to the ones of another row that contains a product that does *not* belong to the current product manager (i.e., change the row number in the parameters accordingly, e.g., from *j_idt35:0:j_idt49* to *j_idt35:1:j_idt49*). Make sure to adapt all names and values so that the parameters indeed correspond to the ones of the other row. Then, forward the request and check (in the browser) whether the product has been removed or not. Even if you have changed the request correctly, the listed products should still be the same, i.e., it was not possible to remove the product of another product manager.

Try to explain why it was not possible to remove the product. You should be able to give a reasonable explanation by using the information you got about the Jakarta Faces ViewState during the lecture. Write your explanation into the following box.

This underlines the positive security benefit of the Jakarta Faces ViewState when used in combination with the *rendered* attribute: It relieves you from having to do programmatic access control checks in the backend because if a button (or anything else that triggers an action that uses a POST request) is not included in the web page (i.e., the form is not rendered), then the user cannot execute the corresponding action – so forced browsing (and in this case a broken object level access control vulnerability) is prevented. When using other frameworks or technologies, this is often not the case, which means that when receiving the request to remove a product (using the example discussed above), you have to check programmatically in the code whether the product truly belongs to the product manager sending the request.

Speaking about access control, let's do another experiment. The code already contains functionality to edit a product. To enable it, you have to do the following:

- Uncomment method *updateProduct* in *AdminProductService.java*.
- Uncomment method *updateProduct* in *EditProductBacking.java*.

- In `view/admin/admin.xhtml`, add another column to the list of products in the same way as you did with the columns that contains the *Remove product* buttons, using the following code:

```
<h:column >
    <h:form rendered="#{request.getRemoteUser() == product.username}">
        <h:commandButton value="Edit product"
            action="#{editProductBacking.editProduct(product)}" />
    </h:form>
</h:column>
```

As a result of this, when logging in as *donald*, the admin area should look as follows:

Code	Description	Price (CHF)	Username		
0001	DVD Life of Brian - used, some scratches but still works	5.95	donald	Remove product	Edit product
0002	Ferrari F50 - red, 43000 km, no accidents	250000.00	luke		
0003	Commodore C64 - used, the best computer ever built	444.95	luke		
0004	Printed Software-Security script - brand new	10.95	donald	Remove product	Edit product

Return to search page Add product Account settings Logout

For the same reasons as above (because of the usage of the *rendered* attribute and the Jakarta Faces ViewState), a product manager only gets *Edit product* buttons in the rows that contain his own products and correspondingly, he can only initiate the editing functionality for his own products.

When clicking an *Edit product* button, the user gets access to `view/admin/product/editproduct.xhtml` to edit the specific product, as illustrated in the next screenshot:

Edit Product

Please modify the following information to edit a product:

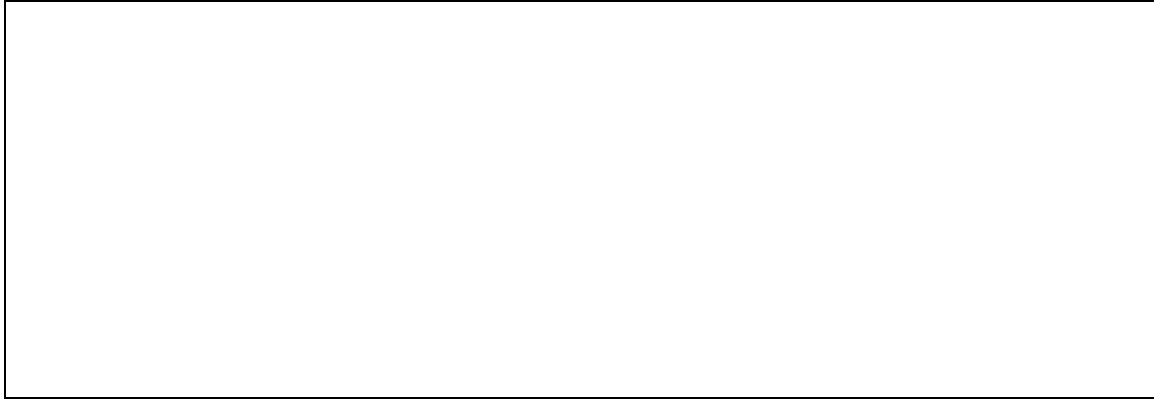
Code:

Description:

Price:

The question here is now whether a malicious product manager can abuse this edit functionality to modify a product not belonging to him. Looking at the code, you can see that when the user clicks the *Update product* button, then method `updateProduct` in `EditProductBacking.java` is called, which itself calls method `updateProduct` in `AdminProductService.java` to update the product in the database. However, neither of these two methods checks whether the product to be updated truly belongs to the current user – so it may indeed be possible that this can be abused to modify products of other product managers. This would again correspond to a broken object level access control vulnerability.

So, is it possible to modify products of other product managers? To answer this, you must analyze in detail the code involved when editing a product and maybe also the request that is sent when clicking the *Update product* button. Hint: This time, the explanation should not be based on the *rendered* attribute or the Jakarta Faces ViewState, as every product manager has access to the form to edit a product. Consequently, there must be another reason why the current implementation is secure or not. Another hint: The annotation `@ConversationScoped` used in `EditProductBacking.java` guarantees that the same backing bean instance is used during the entire edit product process. This scope is used because the other two scopes that are used in most cases – `@RequestScoped` and `@SessionScoped` – are not well suited in this case. Write down your reasoning into the following box.



If you concluded that the current implementation is secure, then you are correct. But it's only secure because the product that is currently edited is kept within the backing bean – and therefore also the primary identifier of the product, the *productID*. This means that a malicious product manager cannot change the *productID* of the product that is currently edited (because he has no access to it), and therefore, he cannot edit products belonging to other product managers. This demonstrates that the backing beans are not just a convenient approach to store state of web pages presented to a user, but that they are also beneficial for security reasons, as critical identifiers do not have to be exposed towards the user (or attacker).

With other frameworks or technologies that do not provide a backing bean-like approach or something similar, the *productID* would have to be exposed towards the user. It is then typically included as a hidden field in the form to edit a product. In this case, a malicious product manager could change the *productID* when submitting the form, so it points to another product. In this case, it would be possible to change products of other product managers, unless it is checked – when the web application receives the request – whether the current user is the owner of the product. Such an approach by exposing the *productID* could also be implemented with Jakarta Faces, but it would be against typical Jakarta Faces best practices and would be a clear indication that the developer didn't understand the Jakarta Faces technology correctly.

7 Task 2: Limit Online Password Guessing Attacks

Right now, an attacker can perform as many login attempts as he likes, which allows online password guessing attack. Note that the usage of PBKDF2 throttles the attacker already to a certain degree, but it is desirable to have a more effective throttling mechanism.

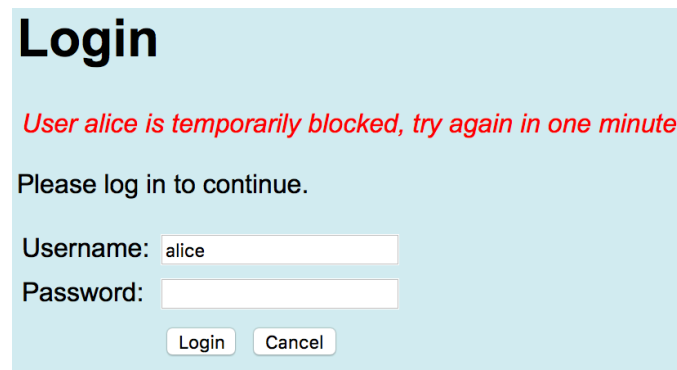
Various throttling mechanisms can be imagined:

- Throttle the attacker after a certain number of failed logins with the same session ID. This defense is not effective because the attacker can easily circumvent this by making sure that a fresh session is established for every attempt.
- Throttle the attacker after a certain number of failed logins from the same IP address. This is effective against weak attackers, but determined attackers can still try large amounts of passwords by performing the logins from a large number of different IP addresses. This would typically be done using a botnet.
- After a certain number of failed logins per username, block login with that username for a certain time. This defense is very effective even against powerful attackers as there is a clear upper bound on the number of login attempts per username and per time interval.

The throttling mechanism you have to implement is specified as follows:

- Whenever a user tries to log in with a wrong password and the used username is currently not blocked, he gets the message *Username or password wrong* (this corresponds to the current behavior of the application).

- After three failed login attempts with a particular username, login with that username is completely blocked for 60 seconds. This is acceptable for legitimate users if they accidentally enter a wrong password three times (which rarely happens) and it does not block users permanently, which could be abused for DoS attacks.
- If a blocked user tries to log in, the message *User xyz is temporarily blocked, try again in one minute* is displayed.



- After 60 seconds, a user can login again, but entering the wrong password (which displays the message *Username or password wrong*) results in blocking the user again for 60 seconds (so when a user is un-blocked after 60 seconds, he gets only one attempt).
- If a user logs in successfully, the application forgets previously false login attempts for this user, i.e., the user gets again three login attempts without being blocked.

To implement the mechanism, you should do the following:

- To keep track of failed logins and blocked users, a service class *LoginThrottlingService.java* is used, of which a skeleton already exists. This class is an EJB with the *@Singleton* annotation, which guarantees there's only one instance in the entire application. In addition, *@Singleton* guarantees thread-safe usage, which means you don't have to deal with synchronized methods. The class provides skeletons of three public methods:
 - *public void loginFailed(String username)*: This method is called to inform that login with *username* has failed. This should increment the login failed counter for that user.
 - *public void loginSuccessful (String username)*: This method is called to inform that login with *username* has succeeded. This should remove all stored information about failed logins of that user.
 - *public boolean isBlocked (String username)*: This method returns whether user *username* is currently blocked.
- First you should complete the implementation of this class. The three methods above are just the public interface of the class; it's up to you how to implement the functionality internally.
- Once the class is implemented, you can inject it in *AuthenticationBacking.java* (which handles the login programmatically²) and use the three public methods described above in the *login* method to limit the number of login attempts per username.

With this mechanism, an attacker is significantly slowed down because even using as many threads, computers, or IP addresses, he can at most perform 1'440 password attempts for a given username per day.

² Note that implementing this login throttling mechanism in Marketplace is only possible because the login is done programmatically, based on *@CustomFormAuthenticationMechanismDefinition*. If we were still using *@FormAuthenticationMechanismDefinition*, then this wouldn't be possible as it wouldn't allow us to influence the actual login. This demonstrated the advantages of *@CustomFormAuthenticationMechanismDefinition* over *@FormAuthenticationMechanismDefinition*.

Take the following implementation hints into account:

- It's quite easy to introduce DoS opportunities with such throttling functionality. Therefore, it is not recommended to wait (actively or passively with *sleep()* or *wait()*) during the blocking time as this may exhaust all available threads. Instead, the web application should return the response immediately to the client in case a login attempt is blocked.
- Implement the mechanism as lightweight as possible so it consumes only little resources. In particular, don't use the database to keep track of failed logins but implement everything «in memory» in the class *LoginThrottlingService.java*.
- Do only handle existing usernames. If a user logs in with a non-existing username, ignore this in your blocking mechanism (don't keep track of failed logins in this case and always display the message *Username or password wrong*). Otherwise, the attacker could exhaust your tracking mechanism by submitting large amounts of fictitious usernames.
- To check the correctness of your approach, you can print the internal State of *LoginThrottlingService.java* whenever a user logs in (successfully or not) by using *System.out.print()*. The output should be visible in the Payara Server output window in NetBeans.

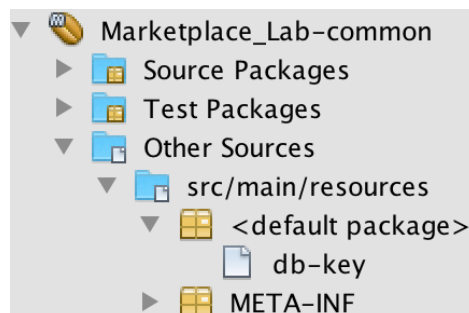
Before you continue, check whether the applicable tests in the lab points section work as expected.

8 Task 3: Encrypt Credit Card Numbers

Currently, the credit card numbers are stored in plaintext in the database. This means an attacker can possibly get access to them via SQL injection, by breaking into the database server, by getting physical access to the database hard disk etc. Likewise, the legitimate (and possibly malicious) database administrator could simply read the credit card numbers directly from the database. To reduce such risks, the application should store credit card numbers only in encrypted form. Within the application, the credit card numbers should of course still be displayed in plaintext.

The mechanism you have to implement is specified as follows:

- AES/GCM/NoPadding with a 128-bit key is used to encrypt the credit card numbers. As GCM (Galois/Counter Mode) is used, a credit card number is not only encrypted, but also integrity-protected (for the sake of simplicity, we will simply say «encrypted» credit card numbers here, but this actually means «encrypted and integrity-protected»). The initialization vector (IV) is stored together with the encrypted credit card number (by concatenating IV and ciphertext). Use a fresh IV for each credit card number. The length of the Auth Tag (used by GCM for integrity-protection) is 128 bits.
- Do not modify the database schema. The current column *CreditCardNumber* in table *Purchase* is a *varchar(100)* field, which is big enough to store the encrypted credit card number. The binary ciphertext (including the IV in front of it) is stored in base64-encoded format.
- The key is stored in a file, in hex-format (e.g., 8E60AF3641D394A104FF356C90AC25B4). This is not perfect but is often done this way in practice. On a productive system, one would configure Payara to run with its own user id and make sure only the Payara user can access that file. The key (*db-key*, don't change it) is already available in a location where it can easily be accessed from within the code:



- To provide the actual encryption and decryption, the service class *AESCipherService.java*³ is used. A static initializer block is already implemented which reads the key from the file system. For encryption and decryption, the methods *encrypt* and *decrypt* are used, for which there exist skeletons.
- JPA provides *AttributeConverters*, which can be used to apply operations on data just before they are written to the database and just after they have been read from the database. We can use this to encrypt the credit card number just before writing it to the database and decrypting it just after reading it from the database. For this, the *AttributeConverter* interface must be implemented. This has already been done with class *AESConverter.java* (in package *ch.zhaw.securitylab.marketplace.common.model*) and you shouldn't adapt this class. Inspecting the two methods in the class, you can see the following:
 - Method *convertToDatabaseColumn* encrypts the data using the *encrypt* method of *AESCipherService* and returns the Base64-encoded encrypted string (which includes the IV and the encrypted credit card).
 - Method *convertToEntityAttribute* decodes the Base64-encoded encrypted data (consisting of IV and encrypted credit card) and decrypts it using the *decrypt* method of *AESCipherService*.

To complete the implementation, you should do the following:

- Implement the methods *encrypt* and *decrypt* in *AESCipherService*. The Javadoc comments of the methods explain the purpose of the parameters and return values and what the methods should do.
- In addition, you must instruct the field *creditCardNumber* in *Purchase.java* to use *AESConverter.class*. This guarantees that the credit card number will be encrypted before it is stored in the database and that it will be decrypted right after reading it from the database. Do this with the annotation *@Convert* as follows:

```
@CreditCardCheck
@Convert(converter = AESConverter.class)
private String creditCardNumber;
```

- To encrypt the credit card numbers of the already existing purchases in table *Purchase*, execute *Marketplace_UpdateEncryptedCreditCards.sql* in *MySQL Workbench*. This script simply replaces the plaintext credit card numbers with AES-encrypted credit card numbers corresponding to 1111 2222 3333 4444 (same value for all rows).

Before you continue, check whether the applicable tests in the lab points section work as expected.

9 Task 4: RESTful Web Service API

The final task is to extend the Marketplace web service so that it provides a RESTful web service API for product managers. In addition, you are going to make sure that login throttling is also done when the REST API is used.

9.1 RESTful API for Product Managers

To do this, you have to complete the REST class *AdminProductRest.java*, which is available as a skeleton. The class contains three methods *get()*, *post()* and *delete()*, which are used to process GET, POST and DELETE requests. The purpose of the three request types is as follows:

- A GET request is used to get all products. For this, a new data transfer object (DTO) *AdminProductDto.java* is used. This DTO is already available and it sends JSON objects in the following form to the client:

³ As it is not possible to inject CDI beans or EJBs into *AttributeConverters*, we use a «normal» class here. <https://stackoverflow.com/questions/31549783/inject-not-working-in-attributeconverter>.

```
[{"productID": "1", "code": "0001", "description": "DVD Life of Brian - used, some scratches but still works", "price": 5.95, "username": "donald"}, {"productID": "2", "code": "0002", "description": "Ferrari F50 - red, 43000 km, no accidents", "price": 250000.00, "username": "luke"}]
```

The URL to be used by the client is *rest/admin/products*.

- A POST request is used to create a new product. For this, *Product.java* is used as DTO (which has already been used as DTO in the lecture to get the results when searching for products) and a new product is sent to the web service as follows:

```
{"code": "0005", "description": "Super Vulnerability Scanner", "price": 9999.95}
```

The URL to be used by the client is *rest/admin/products*.

- A DELETE request is used to delete a product. Of course, a product manager can only delete the own products, which must be checked by the web service.

The URL to be used by the client is *rest/admin/products/{id}*, where *id* identifies the primary key (*ProductID*) of the product that should be deleted.

To test your REST API, there's a test class available in *Marketplace_Lab-rest-client*. It's an extension of the test class used in the lecture, and it tests the already existing REST API and also your extensions. The tests implement various cases using non-authenticated users, authenticated users, valid input data, invalid input data, and so on. For each test, the result is PASSED or FAILED; PASSED means that the test produced the expected response from the web service. To run the tests, simply run *Test.java* in Project *Marketplace_Lab-rest-client* (as described in Section 2). You can run *Test.java* anytime you want, e.g., whenever you have modified one of the methods to handle REST requests to check whether it works as intended. Make sure to always reset the database before running the tests (using both SQL scripts).

In the following, additional details of the three methods and the steps to implement them are given. Note that you don't have to deal with authentication, as this functionality was already implemented during the lecture and is therefore included in the version you are using. Also, the *security-role* entry for the role *productmanager* has already been added to *web.xml*, so you don't have to do this.

Method *get()* (used to process *GET rest/admin/products*):

- Getting the products can be done in a similar way as getting the purchases in method *get()* in *AdminPurchaseRest.java*. However, the list of products retrieved from the database must be converted to a list of *AdminProductDto* objects before they can be sent to the client. To facilitate this, *AdminProductDto.java* provides a constructor to create an *AdminProductDto* object from a *Product* object.
- Use the *@RolesAllowed* annotation correctly to make sure only users with role *productmanager* are allowed GET access to the resource.
- If a user with a different role than *productmanager* tries to access the resource, he should get a 403 response with JSON content *{"error": "Access denied"}*. Assuming you have used the *@RolesAllowed* annotation correctly (see previous step), this should work out of the box as this was already implemented in general during the lecture and is therefore included in the version you are using.
- If a non-authenticated user or a user with an invalid authentication token tries to access the resource, he should get a 401 response with JSON content *{"error": "Authentication required"}*. Just like above, this is already handled in the version you are using and should work out of the box.

Method *post()* (used to process *POST rest/admin/products*):

- The received *Product* object should be validated before processing it. Several validation annotations have already been inserted into *Product.java* before (see task 1), but for completeness, you should add *@NotNull* annotations in front of attributes *code* and *description* in the same way as in *CheckoutDto.java*. As discussed in the lecture, this makes sure that the request is only processed if

the JSON data in the POST request includes all required elements (because if an element is missing, the corresponding attribute won't be set in the *Product* object; i.e., it will be null, which will be detected by the *@NotNull* annotation). Next, you have to add the correct annotation in front of the parameter *product* of method *post()* to make sure that the validation annotations in *Product.java* are enforced.

- Make sure to set the username of the current user in the received *Product* before inserting it into the database. The comment in the method explains how you can get this username.
- To actually insert the entity *Product* into the database, use the *insertProduct* method in *AdminProductService.java* you implemented above (see task 1).
- If the *insertProduct* method of *AdminProductService.java* returns false, a 400 response with JSON content `{"error": "The product could not be added as a product with the same code already exists"}` should be sent to the client. You can do this by throwing an *InvalidParameterException* in method *post()* with the desired error message. The processing of the exception and the generation of the correct response was already done in the lecture and is therefore included in the version you are using (in *InvalidParameterExceptionMapper.java*).
- If a validation error occurs, a 400 response with JSON data containing the validation error message(s) should be sent to the client, e.g., `{"error": "Please insert a valid code (4 letters / digits), Please insert a valid price (between 0 and 999'999.99, with at most two decimal places)"}`. Assuming you inserted the validation annotations correctly in *Product.java* when solving task 1, this should work without further adaptations as handling the validation exceptions was also already implemented in the lecture (in *ConstraintViolationExceptionMapper.java*).
- Use the *@RolesAllowed* annotation correctly to make sure only users with role *productmanager* are allowed POST access to the resource.
- If a non-authenticated user or a user with a role other than *productmanager* tries to access the resource, the behavior should be the same as with GET requests above and this should work out of the box (assuming you have used the *@RolesAllowed* annotation correctly).

Method *delete()* (used to process *DELETE rest/admin/products/{id}*):

- This should be done in a similar way as method *delete()* in *AdminPurchaseRest.java*.
- Use validation annotations with the method parameter *id* to make sure the *id* is between 1 and 999'999 (just like in *AdminPurchaseRest.java*). The message should be *The ProductID must be between 1 and 999'999*.
- Make sure that users can only delete their own products. Getting the name of the current user can be done in the same way as in method *post()* above.
 - As a side note: With the Marketplace Jakarta Faces application, checking that a user can only delete his own products was not necessary when the user clicked on the *Remove product* button, because this was prevented «by design» due to the Jakarta Faces ViewState and by using the *rendered* attribute correctly (see discussion in Section 6.4). But here with the stateless RESTful API, where no «helpful state» is stored on the server side, this must explicitly be checked when the web service receives the request to delete a product (and with many other web application frameworks and technologies, this would also have to be done in this way).
- If an *id* is used that is not in the permitted range, a 400 response with JSON content `{"error": "The ProductID must be between 1 and 999'999"}` should be sent to the client. Assuming you used the validation annotations correctly with parameter *id* (see above), this should work without further adaptations as handling the validation exceptions was already implemented in the lecture (in *ConstraintViolationExceptionMapper.java*).
- If an *id* is used for which no product exists, a 400 response with JSON content `{"error": "The product with ProductID = '999' does not exist"}` should be sent to the client. You can do this by

throwing an *InvalidParameterException* in method *delete()* with the desired error message. The processing of the exception and the generation of the correct response is also already handled correctly (in *InvalidParameterExceptionMapper.java*).

- If a user tries to delete a product that does not belong to him, a 403 response with JSON content `{"error": "Access denied, only the own products can be deleted"}` should be sent to the client. You can do this by throwing an *AuthorizationException* (this custom exception class is already completely implemented) in method *delete()* with the desired error message. To handle this exception, you have to implement an *AuthorizationExceptionMapper*, which can be done similar as in *InvalidParameterExceptionMapper.java*. With *AuthorizationExceptions*, however, the response should use HTTP status 403 (*Status.FORBIDDEN*) instead of 400 (*Status.BAD_REQUEST*). Also make sure the new exception mapper is inserted in *ApplicationConfig.java* (this may happen automatically when building the project *Marketplace_Lab-rest*, but check nevertheless).
- Use the *@RolesAllowed* annotation correctly to make sure only users with role *productmanager* are allowed DELETE access to the resource.
- If a non-authenticated user or a user with a role other than *productmanager* tries to access the resource, the behavior should be the same as with GET requests above and this should work out of the box (assuming you have used the *@RolesAllowed* annotation correctly).

Before you continue, check the following:

- When using *Test.java*, all tests should show PASSED (except the login throttling tests, as login throttling has not been implemented yet for the RESTful API).

9.2 Limit Online Password Guessing Attacks

Finally, make sure that the RESTful API cannot be abused for online password guessing. As you have implemented *LoginThrottlingService.java* above (see task 2), not much remains to be done:

- In the POST method in *AuthenticationRest.java*, use *LoginThrottlingService.java* in a similar way as you have done in the *login* method in *AuthenticationBacking.java*.
- The rules are the same: A username is blocked for 60 seconds after three wrong passwords and after these 60 seconds, the user gets one attempt to log in successfully before the username is blocked again. Just like in the web application, ignore non-existing usernames in your blocking mechanism.
- If the user uses a wrong username or password when the username is not blocked, a 400 response with JSON content `{"error": "Username or password wrong"}` should be sent to the client (this is the current behavior of the web service). The corresponding *InvalidParameterExceptions* are already thrown in the POST method in *AuthenticationRest.java* and are already handled correctly in your version (in *InvalidParameterExceptionMapper.java*).
- If a user is blocked, a 400 response with JSON content `{"error": "User xyz is temporarily blocked, try again in one minute"}` should be sent to the client. You can do this by throwing an *InvalidParameterException* in the POST method with the desired error message. The processing of the exception and the generation of the correct response is already handled correctly (in *InvalidParameterExceptionMapper.java*).

Before you continue, check the following:

- When using *Test.java*, all tests should show PASSED.

Lab Points

In this lab, you can get **6 Lab Points**. To get them, you must demonstrate that your extended Market-place application runs according to the specifications and passes the tests listed in the table below.

- You get 2 points for solving *Task 1: Extending Admin Area* (see Section 6).
- You get 1 point for solving *Task 2: Limit Online Password Guessing Attacks* (see Section 7).
- You get 1 point for solving *Task 3: Encrypt Credit Card Numbers* (see Section 8).
- You get 2 points for solving *Task 4: RESTful Web Service API* (see Section 9).

To get the points, your application must pass the following tests:

Test	Passed
Task 1: Extending Admin Area	
If a non-authenticated user clicks the <i>Admin area</i> button to access <i>view/admin/admin.xhtml</i> , the user is redirected to the login page.	
Logging in with <i>alice/rabbit</i> shows the purchases.	
Logging in with <i>bob/patrick</i> shows an «empty» <i>Admin area</i> .	
Logging in with <i>donald/daisy</i> shows the products and the <i>Add product</i> button.	
The <i>Remove product</i> button is only visible in the rows that correspond to <i>donald</i> 's products.	
Removing a product as user <i>donald</i> removes the product.	
When <i>donald</i> enters a new product, the validation rules work correctly, and appropriate error messages are shown.	
Creating a new product as <i>donald</i> and entering an already existing code does not work and shows an error message.	
Accessing <i>view/admin/product/addproduct.xhtml</i> over HTTP (port 8080) and as user <i>donald</i> results in a redirection to HTTPS.	
Accessing <i>view/admin/product/addproduct.xhtml</i> as user <i>alice</i> results in a 403 error.	
Changing the password of a user works and is designed in a secure way.	
If a non-authenticated user accesses <i>view/admin/account/accountsettings.xhtml</i> , the user is redirected to the login page.	
Task 2: Limit Online Password Guessing Attacks	
Log in with the same user four times in a row using a wrong password. Three times, you should get the message <i>Username or password wrong</i> . After the fourth login attempt, you should get the message <i>User xyz is temporarily blocked, try again in one minute</i> .	
Log in right again with the same user, but using the correct password. The user should still be blocked and get the message <i>User xyz is temporarily blocked, try again in one minute</i> .	
Wait more than 60 seconds and log in again with the same user, with a wrong password. You should get the message <i>Username or password wrong</i> .	
Log in right again with the same user, but using the correct password. You should get the message <i>User xyz is temporarily blocked, try again in one minute</i> .	
Wait more than 60 seconds and log in again with the same user, with the correct password. The login should be successful	
Log in four times (or more) with a non-existing username. You should get the message	

<i>Username or password wrong every time, but never the message that the user is blocked.</i>	
Task 3: Encrypt Credit Card Numbers	
<p>Make two purchases using the credit card number 1111 2222 3333 4444 both times. Then, check with MySQL Workbench that credit card numbers are stored in encrypted form in the database. The entries should look similar as below (the individual entries should all be different due to different IVs):</p> <p>CreditCardNumber</p> <pre>fGd10a46VT4C99RTtDJZqG0XRa8/oPEIDBUcMh3+Qs4qFHS59+DU1CgLS1dqGZZ61+Yp /DKpVnIBMtDs7Rr7dh/9rf+EDUB03NWAX9gNAkGurizP+lzQ/1a/rZkvCDaXuNr26a1k U3lEXINeH7uKvjJSmYnfD2e0/nDSKRsWDLDDYw89dqW5fEOGb4isz4l51yq9Yx88w62y</pre>	
Accessing the admin area as user <i>alice</i> (or <i>robin</i>) shows the credit card numbers in plaintext.	
Task 4: RESTful Web Service API	
When running <i>Test.java</i> in project <i>Marketplace_Lab-rest-client</i> , all tests show PASSED.	

Handing in your solution can be done in two different ways:

- Demonstrate that your extended Marketplace application passes all the tests to the instructor in classroom, during the official lab lessons.
- Make a video that shows that your extended Marketplace applications passes all the tests. Send the video by e-mail to the instructor. Use *SecLab - Marketplace - group X - name1 name2* as the e-mail subject, corresponding to your group number and the names of the group members. If the video is too big for an e-mail, place it somewhere (e.g., SWITCHdrive) and include a link to it in the e-mail.